Evaluating the Applicability of OpenISBT to Open Source Microservice APIs

Amit Jerochim

amit.jerochim@campus.tu-berlin.de

December 07, 2020

BACHELOR'S THESIS

Mobile Cloud Computing Research Group

Technische Universität Berlin

Examiner 1: Prof. Dr.-Ing. David Bermbach

Examiner 2: Prof. Dr.-Ing. Stefan Tai

Advisor: Martin Grambow

Declaration

I hereby declare that I have created the present work independently and by my own without illicit assistance and only utilizing the listed sources and tools.

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Die selbständige und eigenständige Anfertigung versichert an Eides statt:

December 07, 2020

Amit Jerochim

Abstract

Benchmarking serves to improve the non-functional properties of IT-systems. Designing a portable benchmark for microservices is difficult as they do not share a common interface. In this work, we evaluate the applicability of openISBT, a proof-of-concept prototype of a portable and pattern-based benchmarking approach for REST-based microservices, described in a machine-understandable way. OpenISBT allows developers to model interaction patterns as sequences of abstract operations and define abstract workload on them. It maps specific service operations to abstract operations using matching units and generates service-specific workload. We determine the coverage of service operations and APIs that openISBT can map and determine abstract operations' distribution for samples of more than a thousand interface descriptions. We also derive new abstract operations from a dataset of interface descriptions, implement new matching units, and contribute to existing matching units. Therefore, we implement a web crawler to collect open source interface descriptions and a prototype to measure the coverage metrics. Our work shows that openISBT supports 41.4% of the APIs and 75.2% of the service operation in the dataset. Our contributions to openISBT increase the coverage of supported APIs to 48.8% and the coverage of supported services operations to 82.3%.

Zusammenfassung

Benchmarks dienen der Verbesserung nichtfunktionaler Eigenschaften von IT-Systemen. Der Entwurf eines portablen Benchmarks für Microservices ist jedoch problematisch, da diese keine Standardschnittstelle teilen. In dieser Arbeit untersuchen wir die Anwendbarkeit von openISBT, einen Prototypen eines auf Interaktionsmustern basierenden Ansatzes, um Microservices mittels Benchmarks zu messen. Diese müssen REST-basiert sein und es muss für sie eine Schnittstellenbeschreibung existieren. OpenISBT erlaubt Entwicklern Interaktionsmuster als Sequenzen abstrakter Operationen zu modellieren und abstrakte Last für jene Interaktionsmuster zu definieren. Der Prototyp bildet spezifische Services auf abstrakte Operationen ab und kann so service-spezifische Last generieren. Wir bestimmen anhand verschiedener Stichproben mit je einem Umfang von über eintausend Schnittstellenbeschreibungen die Abdeckung von service-spezifischen Operationen und Schnittstellen, die openISBT abbilden kann. Außerdem bestimmen wir die Verteilung service-spezifischer Operationen auf die abstrakten Operationen. Darüber hinaus leiten wir aus dem Datensatz neue abstrakte Operationen ab und entwerfen und implementieren Komponenten, die service-spezifische Operationen auf die neuen abstrakten Operationen abbilden. Wir verbessern außerdem die Funktionalität bestehender Komponenten. Hierfür entwickeln wir jeweils einen Prototypen, um Schnittstellenbeschreibungen zu sammeln und um Abdeckungsmetriken zu bestimmen. Unsere Arbeit zeigt, dass openISBT 41.4% der untersuchten Schnittstellen voll unterstützt und 75.2% der service-spezifischen Operationen unterstützt. Durch unsere Erweiterungen steigerten sich die Abdeckung der voll unterstützten Schnittstellen auf 48.8% und die Abdeckung der service-spezifischen Operationen auf 82.3%.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Microservices architectural style steadily replaces monolithic architectures to improve development processes and maintenance of large and distributed software applications [16, 38]. In this work, we evaluate the applicability of openISBT, a pattern-based benchmarking framework designed for microservices. This chapter presents the problems we try to solve and the goals we try to achieve. Also, it conveys an overview of this thesis's structure.

1.1 Motivation & Problem Description

When designing and implementing services, continuous quality improvement plays a central role [5, p. 28]. Herein quality describes how good a service is at a specific ability, e.g., how secure or scalable[5, p. 17]. Therefore, the service provider can rely on benchmarking techniques to ensure quality improvement. Usually, a benchmark has to be designed for every service and maintained with every service version [20, sec. 1]. The pattern-based benchmarking approach enables the reuse of benchmarks [20]. However, there are still open questions:

1. The approach has only been evaluated with a few toy services. It is unknown whether this approach is applicable for all REST compliant microservices in production and, if not, the proportion of services it is applicable for is unknown.

2. The authors also only define abstract operations that map to CRUD operations. Some APIs implement uncovered service operations such as GET /validatePhone, GET /login, or GET /purchase, which openISBT cannot map to the currently defined abstract operations. We assume more abstract operations, such as methods for authentication, validation, or triggering business processes. It is unknown whether real-world microservices require additional abstract operations and how frequently openISBT would map them.

1.2 Goals of the Thesis

This thesis aims to evaluate and extend the applicability of openISBT to open source microservices APIs. To achieve our goals, we will make the following contributions.

1. Implementing a web crawler and collecting a large set of OpenAPI 3.0 files
2. Designing and implementing a tool to evaluate the applicability of openISBT for the set of collected OpenAPI 3.0 files
3. Quantifying the applicability of openISBT
4. Deriving new abstract operations and implementing corresponding matching units based on real-world service operations that openISBT does not support yet
5. Quantifying the improvement of the applicability of openISBT through the new abstract operations

1.3 Structure of the Thesis

Our thesis is divided into ten chapters, whereby this introduction represents the first chapter. It conveys a brief overview of the area of benchmarking REST-based microservices and motivates the pattern-based benchmarking approach.

Our work is based on theoretical concepts and technologies. The second chapter introduces all these concepts and conveys all the knowledge required to understand our work.

Chapters three to seven are the main part of our work and describe the experiment and the results. Chapter three gives a general overview of the experiment, and the subsequent chapters convey a detailed view of the single steps of the experiment and the results.

The last three chapters serve to reflect our own doing. In chapter 8 we analyze methods used and assumptions made, we discuss limitations of openISBT and our prototypes, and finally introduce open questions and suggestions for future research. In chapter 9 we compare our work to other research and discuss the advantages and disadvantages of the different approaches. In chapter 10, we finally conclude the experiment and its results.

Chapter 2

Background

This chapter conveys the general background knowledge required to understand the addressed problems and ways we solved them.

2.1 Microservices

The term microservices describes an architectural style where complex application systems are split into multiple independent, loosely coupled, and deployable services built around business capabilities, which communicate using lightweight mechanisms. There is no clear definition for the microservices architectural style, but there are common characteristics they usually share [16]. However, for clients (such as openISBT), most of these characteristics concern hidden aspects, so they are less relevant in this work. For example, decentralized data management is crucial for microservices, although the client cannot observe if multiple services share the same data storage or not.

The communication between services is, on the other hand, highly relevant in this work. The term ”smart endpoints and dumb pipes” [16] describes an approach where complexity is, if

required, added to the service itself in order to keep the communication mechanisms as simple as possible. Therefore, the microservices architectural style avoids complex middleware systems such as Enterprise Service Bus [9] and moves functionality such as business logic or routing to the service itself. Microservices use either REST and HTTP or lightweight message brokers to communicate with each other [16]. RabbitMQ[1] and ZeroMQ[2] are representants for lightweight message brokers, which use the AMQP[3] protocol. However, openISBT can only benchmark REST-based and HTTP-based microservices [20, sec. 5], so we focus on them in this work. Section 2.3 discusses REST and HTTP in detail.

2.2 Benchmarking

An important aspect when providing services is their continuous quality improvement [5, p. 28]. Especially if we consider that microservices' implementations regularly fluctuate, then measuring quality change is crucial.

Each kind of quality describes how good a system is at doing something. Therefore, qualities are the non-functional properties of a system [5, p. 17]. In contrast, the functional properties of a system describe what the system does and not how. Quality itself has many faces, such as availability, performance, security, reliability, scalability, correctness, or cost-efficiency [5, p. 19-22]. Improving one kind of quality may affect other types negatively. For example, increasing security might decrease performance because of additional calculations [5, p. 24].

Benchmarking is an established approach to measure a system's qualities. When we benchmark a system, we actively put external pressure on it using a workload generator and measure how it can handle this pressure. Monitoring is an alternative approach for measuring a system's qualities. In contradiction to benchmarking, it describes a passive observation of a

[1] https://www.rabbitmq.com/
[2] https://zeromq.org
[3] https://www.amqp.org/

system that is already in production [5, p. 24]. Benchmarking, however, allows evaluating the system in a controlled experiment before the system is deployed to production, which is crucial for simulating critical situations (e.g., user traffic of an online-shop on Black Friday).

Especially from the client's perspective, the SUT is a black box, and information about the SUT is not available [5, p. 11-12]. For instance, a microservice's implementation is hidden behind the facade of HTTP.

"Benchmarking in general follows a three-step process of benchmark design, benchmark execution, and dealing with benchmark results" [5, p. 14-16], whereas the benchmark execution covers aspects regarding the benchmark implementation and experimentation. In later chapters and sections, we outline and discuss specific design and implementation objectives that concern our work.

2.3 Representational State Transfer and the Hypertext Transfer Protocol

Representational State Transfer (REST) [15], the de facto standard when building APIs for microservices [16], is an architectural style for network-based software and mostly used over the Hypertext Transfer Protocol (HTTP). A set of constraints defines this architectural style. For example, client-server architectural style [15, sec. 5.1.2], stateless communication between server and client [15, sec. 5.1.3], cache [15, sec. 5.1.4], uniform interface [15, sec. 5.1.5], layered system [15, sec. 5.1.6], and optionally code-on-demand [15, sec. 5.1.7]. The uniform interface is the most important constraint and covers four subordinated interface constraints.

REST uses the concept of resources to abstract information [15, sec. 5.2.1.1]. The interface constraint **resource identification in requests** forces the use of resource identifiers such as URI to name resources and representations to describe them [15, sec. 5.2.1.2]. Representa-

tions might differ in their media types (e.g., XML, HTML, or JSON), so clients might prefer different representations. Therefore, the constraint **resource manipulation through representations and content negotiation** is required [15, sec. 5.2.1.2]. Content negotiation means that the client and server should agree on a representation. Also, a representation should contain all required data to modify the resource's state. Next, REST forces messages to be **self-descriptive**. It means that a request contains all information required by the server to process the message. This includes the use of standard methods and media types to indicate semantics [15, sec. 5.3.1]. For HTTP, this also means that the HTTP method chosen should fit the best to the request [15, sec. 6.3.3.2]. Finally, representations should contain hypermedia links that indicate possibilities for the client on how to proceed. This is also known as **Hypermedia as the engine of application state (HATEOAS)** [15, sec. 5.2.1].

Richardson's maturity model (RMM) [17] breaks down the principles of REST into three compliance levels. An application that violates all interface constraints is not compliant with REST at all. In order to follow the first level of RMM, the application should introduce resources and use resource identifiers. If requests are also self-descriptive, then the application also follows level two of RMM. Therefore, the second level of RMM forces the use of HTTP methods that fit the best. For example, if the client does not want to change the server's state, it should use HTTP GET. The third level of RMM also forces HATEOAS. Roy Fielding clarifies that the third level of RMM is crucial for an API to be RESTful [14]. However, APIs that follow the second level of RMM are sufficient for the pattern-based approach.

2.4 Interface Description

An application programming interface (API) defines interactions between multiple software components on a source code level. If components do not understand each other's source code, then they first have to agree on a common language. A language that describes interfaces is

called interface description language (IDL). There are different language-independent standards, such as Web Services Description Language[4], Android Interface Definition Language[5], or COBRA IDL[6].

There are also common interface description languages for REST-based and HTTP-based APIs, such as RAML[7], API Blueprint[8], or OpenAPI[9]. In this work, we focus on the OpenAPI Specification (OAS). It allows describing services without access to the source code or inspection of the network traffic. A document describing a specific API is called an OpenAPI document and is both human-readable and machine-readable. It also contains all information required to call any service operation and to process its response. We also use different tools that are available around the specification, such as SwaggerHub, a collaborative platform for designing and documenting APIs, or the Swagger Editor, a dedicated editor for OpenAPI documents with an integrated validation functionality.

The content of an OpenAPI document is separable from its syntax. The specification allows the use of either YAML or JSON to describe the same content. As openISBT can only handle JSON formatted OpenAPI documents, we only use JSON files in our work. However, since YAML is better readable for humans than JSON, we use mainly YAML to illustrate examples.

In chapter 9, we discuss the potential extendability of openISBT for handling AsyncAPI[10]. This IDL was discussed in the last year to become an industry standard [36] and can be used for microservices based on RabbitMQ or other asynchronous APIs.

[4] https://www.w3.org/TR/2007/REC-wsdl20-20070626/
[5] https://developer.android.com/guide/components/aidl
[6] https://www.corba.org/omg_idl.htm
[7] https://raml.org/
[8] https://apiblueprint.org/
[9] https://swagger.io/specification/
[10] https://www.asyncapi.com/

2.5 Pattern-based Benchmarking Approach

There are several benchmarking approaches for systems that share a common interface (e.g., databases) [2, 6, 8, 10, 11, 27, 32, 33, 40–43]. A portable benchmarking tool does not exist for microservices yet, as their requests can vary broadly, and they do not share a common interface.

Martin Grambow et al., however, argue that a benchmark should rely on the REST architectural style and present a pattern-based benchmarking approach and openISBT, a proof-of-concept prototype [20]. The second level of RMM supposes the use of HTTP methods and media types that describe service operations as well as possible. Martin Grambow et al. introduce the concept of abstract operations to design a portable benchmarking tool. The authors define abstract operations with a specific purpose. They argue that REST-based service operations with similar purposes also share the same HTTP methods and media types. Therefore, openISBT should match specific service operations to these abstract operations.

The authors present the five abstract operations CREATE, READ, SCAN, UPDATE, and DELETE that describe CRUD operations. SCAN and READ are both reading operations, which do not change the server's state and differ only in the resource scheme. SCAN describes service operations where the client receives a collection of resources, and READ describes services where the client receives a single resource.

As benchmarks should also be realistic, the workload is not defined on single abstract operations but on sequences of abstract operations called interaction patterns. The approach allows developers to configure custom interaction patterns. To provide an example, we assume an online-shop that offers different products. A client might search for a product category and get a list of products. Then, it could pick a specific product from the list to get a detailed view. Therefore, it would first perform GET /products. Assuming the server would respond with a collection of products and one of them has the ID 1, the client could perform GET /products/1 to retrieve additional information about the product. We can define an abstract

interaction pattern to describe this interaction. First, the client performs a SCAN, and then it performs a READ.

The binding follows a three-step process. First, openISBT checks for each endpoint if it defines all operations to execute the interaction we define in the pattern. For the example above, it matches GET /products to SCAN and GET /products/id to READ. Assuming, the online-shop also allows retrieving a customers' list by performing GET /customers and creating a new customer by performing POST /customers. In this case, openISBT would match GET /customers to SCAN but could not match READ. Therefore, this endpoint is not supported, and openISBT cannot benchmark it with this interaction pattern. If openISBT can match all operations, it resolves dependencies between services. The operation GET /products/id defines a path parameter, and openISBT should replace it with a value that it receives after performing GET /products. Finally, openISBT ensures executability by checking if dependencies could be resolved. If executability is granted, openISBT generates a service-specific workload.

In our work, we focus on the first step and analyze each abstract operation isolated.

Chapter 3

Overview

To analyze and extend the applicability of openISBT, we construct an experiment. Figure 3.1 shows the experiment design. In this experiment, we first collect a set of OpenAPI documents from SwaggerHub. Next, we use an evaluation tool to determine the applicability of openISBT. Afterward, we derive new abstract operations and extend openISBT. Therefore, we analyze all service operations that openISBT cannot match, cluster them into groups of operations with similar purposes, and analyze common properties for each group. We design and implement new matching units to match service operations to the new abstract operations based on these common properties. Finally, we repeat the measurement and compare the results of both measurements.

Computer programs automate both the data collection process and the measurements described above. Therefore, we require preceding activities that the experiment design in figure 3.1 does not reveal. We design and implement a web crawler to collect OpenAPI documents and a tool that executes the measurements. The tool also integrates the components for data cleaning. Chapter 4 presents the web crawler's design and implementation and several components for data cleaning. Chapter 5 discusses aspects regarding the measurement design and measurement execution. It presents the evaluation tool's design and architecture

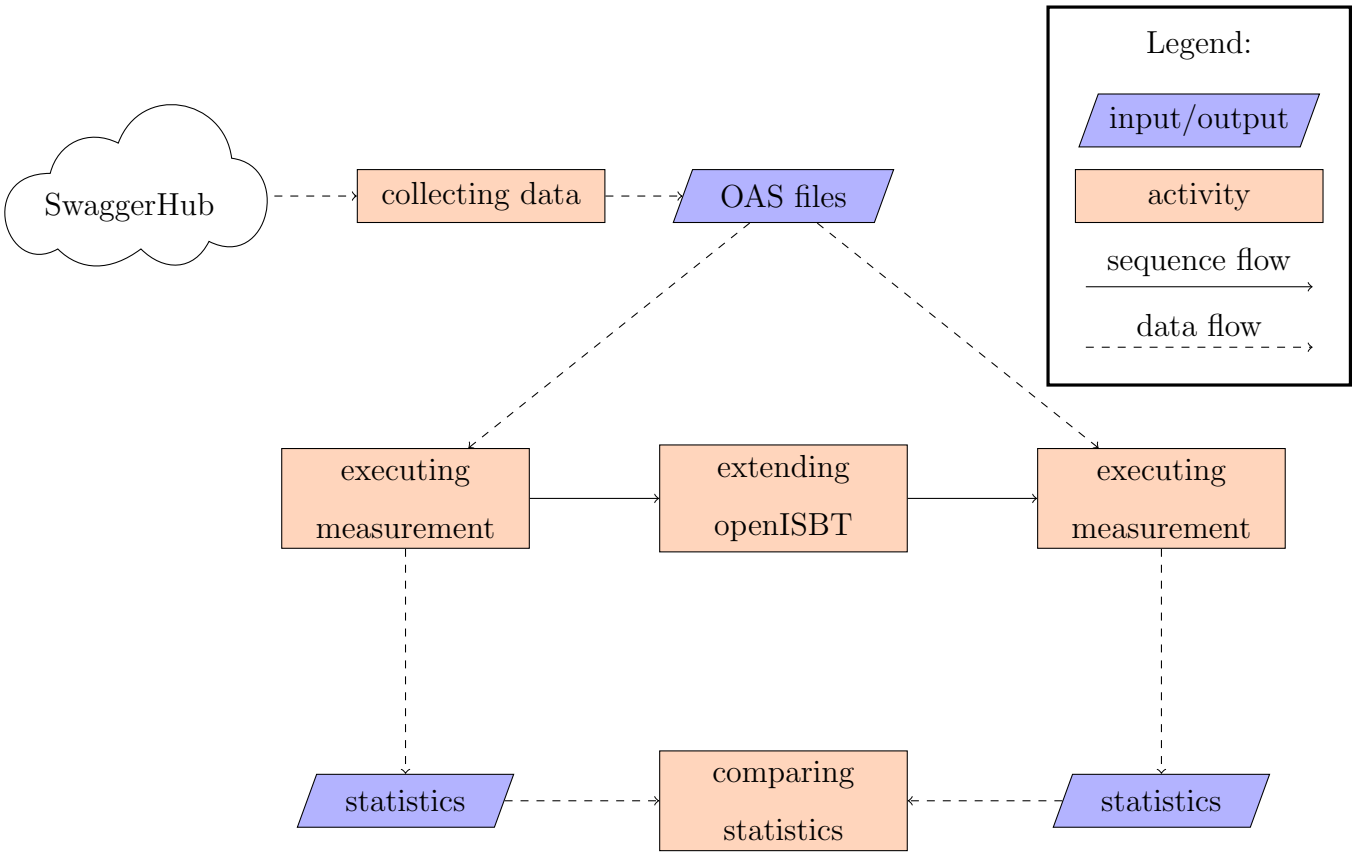


Figure 3.1: General structure of the experiment

and discusses the first measurement results. Chapter 6 discusses data findings, presents new abstract operations and contributions to openISBT. Finally, chapter 7 presents the results of the second measurement. It compares both measurements' results and highlights the achieved improvement.

Chapter 4

Collecting OpenAPI Documents

Since openISBT has only been evaluated with a few toy services, the first goal is to evaluate its applicability with a large number of services used in production.

Therefore, we first require a set of OpenAPI documents, so we have to collect and clean them. Both steps, the collecting as well as the cleaning of the data, happen fully automated. Therefore, in this chapter, we explain the design and functionality of different tools we implement to collect and clean the data. In section 4.1 we present the design of our web crawler. In section 4.2 we introduce different cleaning criteria and explain why subsequent data cleaning is necessary even if the web crawler supplies only valid OAS 3.0 documents. We also explain why we integrate our cleaning modules as part of the evaluation tool and clean the data at runtime during each analysis.

4.1 Designing a Web Crawler

Understandably, a web crawler cannot stand for itself and requires a website to crawl data from. The company SmartBear offers a SaaS solution for service providers to design and

maintain APIs. It also allows service providers to manage, host, and distribute interface descriptions of their APIs. As this tool has a web interface designed for humans, we implement a web crawler to automate repeatable user interactions and collect 2069 OpenAPI documents. The web crawler proceeds with four steps in order to collect the documents:

1. It creates a WebDriver instance and configures it.
2. Afterward, it searches for APIs performing HTTP requests and saves responses as HTML files.
3. Then it reduces the HTML files to a single file containing all URLs we want to request.
4. Finally, it checks for each API whether its API description is valid and, if so, it downloads it.

4.1.1 Configuring the WebDriver Instance

The first and also a relatively short step is the creation and configuration of the browser instance. For this, we use the Java WebDriver library[1] provided by Selenium and geckodriver[2] provided by Mozilla. The configuration is done by setting the window size of the WebDriver, i.e., the browser, to full screen. This step is crucial as we use awt.Robot[3], which interprets coordinates as display coordinates, while the WebDriver API interprets them as browser coordinates. Therefore, setting the window size to full screen makes both APIs interpret coordinates as the same position on the display.

[1] https://www.selenium.dev/selenium/docs/api/java/
[2] https://github.com/mozilla/geckodriver
[3] https://docs.oracle.com/javase/7/docs/api/java/awt/Robot.html

4.1.2 Searching for APIs

After the WebDriver is instantiated and configured, the crawler uses the search functionality of SwaggerHub to get an overview of existing documents. The website also allows refining the search by adding different filter parameters such as type, owner, or a Query [39].

From the client's perspective, the search is an HTTP GET request, i.e., the client performs a GET /search request to the server and refines the requested results by adding the filter parameters as query strings to the path. For instance, to search only for public documents, the HTTP request should be GET /search?visibility=PUBLIC. This example implicitly hints that the values of the query strings are case sensitive. They should be written in capital letters to be processed correctly.

For our experiment, we require open source documents used in production in OAS 3.0 format. The example above already filters only public, i.e., open source documents, and we extend it by the parameters specification, type, and state. Table 4.1 shows an overview of all filter parameters we use, including allowed values and explanations. We set the specification to OAS 3.0. The type parameter allows us to search for APIs only and reject libraries for common components, e.g., a complex data model used in many APIs. Also, we search for published APIs only as we assume that a service provider would not rely on unstable and mutable APIs for services in production.

Table 4.1: Search syntax for SwaggerHub

Parameter	Possible Values	Explanation
Specification	• OPENAPI3.0 • SWAGGER2.0	SwaggerHub hosts OpenAPI 2.0 formatted documents (formerly known as Swagger 2.0) and OpenAPI 3.0 formatted documents. Using this parameter, we can specify the required format.
Visibility	• PUBLIC • PRIVATE	The visibility shows if a document is either private or public. Private means that a document is only visible to its owner and contributors. Public documents are visible to anybody, i.e., also to unregistered users.
State	• PUBLISHED • UNPUBLISHED	The state shows whether an API is ready for use in production or not. Setting the state to published signals that an API is stable and makes it also read-only.
Type	• API • DOMAIN	The type shows whether a document is an API description or a domain. A domain is a library of common components used in multiple API descriptions, e.g., a user object's data model.

Continued on next page

Table 4.1 – continued from previous page

Parameter	Possible Values	Explanation
Page	Any page number starting with one	SwaggerHub uses pagination to limit requested results. The client only gets the first ten items per request. Using this query string parameter, we can request additional items.

The request in 4.1 follows all requirements.

$$GET \quad /search?spec = OPENAPI3.0\&type = API\& \\ state = PUBLISHED\&visibility = PUBLIC \quad (4.1)$$

In fact, the request's response in 4.1 only returns the first ten results that match our pattern, so the request is equal to the request in 4.2.

$$GET \quad /search?spec = OPENAPI3.0\&type = API\& \\ state = PUBLISHED\&visibility = PUBLIC\&page = 1 \quad (4.2)$$

Getting more results affords the performance of multiple requests with different values for the page parameter. However, SwaggerHub does not process those kinds of requests reliably. For most of the requests, it responses with the required data, but too frequently, we receive an HTTP 504 Gateway Timeout. As SwaggerHub practically always responds with HTTP 200 if the request is easier to process, we simplify the query and perform the request in 4.3 instead:

$$GET \quad /search?state = PUBLISHED\&visibility = PUBLIC\& \\ type = API\&page = \langle page\_number \rangle \quad (4.3)$$

Therefore, the results set also contains results we do no require, so we shift logic for filtering to our own program. We conclude this step by saving each result, represented by an HTML document to a file.

4.1.3 Processing the Search Results

In this step, the web crawler reduces all the collected HTML documents to a list of hypertext links to OpenAPI documents it tries to collect. For each hypertext link in each document, it first checks whether the OpenAPI documents have a Swagger 2.0 format or OpenAPI 3.0 format. If the format is OpenAPI 3.0, the web crawler persists the hypertext link.

To implement it, we make use of jsoup[4], a library that allows us to efficiently work with HTML files and make use of CSS selectors. The web crawler uses jsoup to parse each HTML document to a DOM. Then it uses CSS selectors to filter all Content Division Elements, i.e., $\langle div \rangle$-container, to a list of HTML-Elements, each containing meta-information about an OpenAPI document such as the hypertext link, the specification, or the owner. It rejects all elements if the format is not OpenAPI 3.0 and persists the hypertext links in the remaining elements in a text file.

[4] https://jsoup.org/

4.1.4 Saving the OpenAPI Documents

Finally, the web crawler downloads all valid OpenAPI documents. For each document, it first opens the hypertext link, verifies the document's correctness, and finally downloads a JSON representation of the API description document. Again, we use the WebDriver library to request each URL. However, if humans visit any of these pages, they interact with a dynamic website, and the DOM structure depends on users' interactions. We use the awt.Robot library again to trigger mouse and keyboard events such as mouse move, hover, or click. To find DOM elements use CSS selectors provided by the WebDriver library.

To verify the correctness of an OpenAPI document, we rely on SwaggerHubs' own validation. The information, if a document is valid or not, is shown on the page and is therefore also part of the DOM. The web crawler uses CSS selectors to find this information. If a document is valid, the web crawler downloads it. The download hyperlink is not a permanent part of the DOM, so the web crawler navigates through the page as a real user would do in order to trigger different events and change the DOM structure.

4.2 Cleaning the Data

In the previous section, we mention that the web crawler downloads only valid documents. Nonetheless, additional data cleaning is necessary. Usually, the term data cleaning refers to the process of detecting and correcting or removing corrupt data [12]. In this paper, we defiantly define the term as the decision to include a document in our statistics. As we pass a set of documents to our evaluation tool, we use the term of discarding a document from that set as the decision to ignore it when generating the statistics.

In this section, we introduce different cases where discarding documents is necessary or at least recommendable. We explain why we decided to implement the data cleaning modules as part of our evaluation tool and discard unwanted documents at runtime when running

each analysis.

In some cases, we postpone the question of why we discard documents to chapter 5 as we need to introduce some mathematical considerations first.

4.2.1 Discarding Non-production OpenAPI Documents

As we want to analyze services used in production, we have to remove API descriptions of other APIs before doing our analysis, which would result in a manual investigation for each file. As this seems to be infeasible for large sets, we automatically reduce non-production data. We make a hard assumption that short OpenAPI documents are probably not used in production to implement automation. This approach results in a trade-off as we, on the one hand, might discard small APIs used in production but, on the other hand, might keep undesired documents. Using a boundary of 100 LOC, we mostly discard wanted documents, which is tolerable as long as we keep the number of unwanted documents in our sample small.

4.2.2 Discarding Documents That Only Define Operations on Nested Resources

In REST, each resource stands on its own and is addressable behind a URI. Also, the relationship between resources is not defined by their URIs but by their hypermedia links. However, several implementations use the concept of sub-resources in their routing mechanisms. For instance, Rails[5] or JAX-RS[6].

OpenISBT also relies on the concept of sub-resources but can not match services on them

[5] https://guides.rubyonrails.org/routing.html#nested-resources
[6] https://docs.jboss.org/resteasy/docs/1.0.1.GA/userguide/html/JAX-RS_Resource_Locators_and_Sub_Resources.html

yet, so we can not implement an abstract operation that would match them. Chapter 5 shows that keeping documents that only define operations on sub-resources leads to a wrong determination of higher applicability.

4.2.3 Removing Unusual Formatted OpenAPI Documents

Some of the OpenAPI documents do not fit our requirements, so we discard them. In the following, we introduce two kinds of unwanted documents.

Discarding Documents Without a Paths-Object

We require an OpenAPI document containing a paths-object for our analysis. Listing 4.1 shows a minimal example with the required format. Listing 4.2 shows an example that SwaggerHub considers as valid but does not contain a paths-object. Therefore, we check for any OpenAPI document if it contains a paths-object, and if not, we discard it.

```
{
  "openapi" : "3.0.0",
  "info" : {
    "version" : "1.0.0",
    "title" : ""
  },
  "paths" : {...}
  ...
}
```

Listing 4.1: OpenAPI document with the required paths-object

```
{
  "components" : {...}
}
```

Listing 4.2: Valid OpenAPI document without a path-object

Discarding Documents With General Path-Properties

The OpenAPI 3.0 specification offers different ways to describe the same objects. Therefore, different OpenAPI documents can describe the same API.

For example, service operations have different predefined properties [29] and also might have customized properties [31]. If such a property is identical for all service operations on an endpoint, developers can define it once at the endpoint's root [30] instead of repeating code.

Considering this case would add unwanted complexity to our tool but increase the number of documents we can analyze. However, our observations show that most of the public OpenAPI documents only define service operations at the endpoint's root and no other properties. Hence, the added value is not worth adding this complexity.

4.2.4 Removing OpenAPI Documents That OpenISBT Cannot Handle

To evaluate the applicability of openISBT to an arbitrary API, we pass the API description to openISBT and to analyze information that openISBT writes to the standard output. If openISBT cannot process a document without throwing an exception, then the stack trace is written to the standard output. The required output is also incomplete, so we cannot analyze it.

Discarding this document at runtime is crucial, as we make changes to the code of openISBT and then repeat the analysis. Therefore, an arbitrary change to the code can influence the program flow in different ways:

1. An OpenAPI document that makes openISBT throwing an exception before code change still makes it throwing an exception after a code change.

2. An OpenAPI document that openISBT can handle correctly before code change can still be handled correctly after a code change.

3. An OpenAPI document that makes openISBT throwing an exception before code change can be handled correctly after a code change.

4. An OpenAPI document that openISBT can handle correctly before code change makes openISBT throwing an exception after a code change.

For cases (1) and (2), we can discard the documents once in advance without any disadvantages. For case (3), discarding the document in advance would unnecessarily reduce the sample in subsequent analyses. For case (4), discarding the document in advance would lead to wrong results in subsequent analyses, which is critical.

Chapter 5

Analyzing OpenAPI Documents

To evaluate the applicability of openISBT, we require a measurement. Section 5.1 covers considerations and challenges regarding the measurement design and the evaluation tool's design and implementation. In section 5.2, we discuss aspects regarding the measurement execution and the results of the first measurement execution.

5.1 Designing an Evaluation Tool

This section introduces our measurement design and presents a tool that follows our measurement design and executes measurements. The measurement design defines metrics and instructions to analyze data. The tool is required to automate the process and ensure correctness, reproducibility, and understandability.

5.1.1 Deriving Metrics to Quantify the Applicability

To derive metrics, we need to understand REST APIs as sets of service operations.

As we mention in section 2.5, openISBT uses matching units to bind abstract operations to service operations based on their properties. If the matching succeeds, then a service operation is supported, and otherwise, the service operation is not supported.

Given an arbitrary REST API i, which provides a set N_i of service operations. $M_i \subseteq N_i$ is a subset of supported service operations. Then we can state that equation 5.1 shows a suitable metric to describe the coverage of supported service operations for this API.

$$p_i = \frac{|M_i|}{|N_i|} \qquad \text{with} \qquad 0 \leq p_i \leq 1 \tag{5.1}$$

Since we require in a metric to quantify the coverage of supported service operations for a set A of n APIs, we need to extend the metric from equation 5.1. Equation 5.2 shows the extended metric. The numerator $\sum_{i=1}^{n} |M_i|$ describes the sum of all supported operation across all APIs in A. The denominator $\sum_{i=1}^{n} |N_i|$ describes the sum of all supported and unsupported operations across all APIs in A.

$$p_{operations} = \frac{\sum_{i=1}^{n} |M_i|}{\sum_{n=1}^{n} |N_i|} \qquad \text{with} \qquad 0 \leq p_{operations} \leq 1 \tag{5.2}$$

From an API provider's perspective, who considers integrating openISBT in CI pipelines, $p_{operations}$ might not be suitable enough. The API provider requires an estimate if openISBT supports all his service operations, i.e., if $p_i = 1$. Even if $p_{operations}$ is very high, we cannot conclude that an API with $p_i = 1$ exists.

Equation 5.3 describes an alternative metric. Given again a set A of n APIs, then we state for an API i that $M'_i = 1$ if openISBT supports all its service operations, i.e., if $p_i = 1$, or $M'_i = 0$ otherwise. Then the metric p_{APIs} describes the ratio of APIs in A that are fully supported.

$$p_{APIs} = \frac{\sum_{i=1}^{n} M'_i}{|A|} \quad \text{with} \quad M'_i = \begin{cases} 1 & : p_i = 1 \\ 0 & : p_i < 1 \end{cases} \quad \text{and} \quad 0 \leq p_{APIs} \leq 1 \tag{5.3}$$

5.1.2 Requirements and Purposes

The main purpose of the evaluation tool is to calculate $p_{operations}$ and p_{APIs} for a given input of API description files. Algorithm 1 shows instructions to calculate p_i. Appendix A.1 introduces additional algorithms, which use algorithm 1 to calculate p_{APIs} and $p_{operations}$.

Algorithm 1: Calculate coverage metric p_i for a single API i

input : OpenAPI file

input : pattern configuration file

output: p_i

1 ALL_OPERATIONS = Count all operations listed in the OpenAPI file;

2 Try to bind the abstract interaction patterns to service operations in the OpenAPI file;

3 SUPPORTED_OPERATIONS = Count the supported operations;

4 p_i = SUPPORTED_OPERATIONS divided by ALL_OPERATIONS

The mathematical considerations behind algorithm 1 are not very challenging. Nonetheless, the algorithm introduces a set of undiscussed issues:

1. The tool should handle every set of files, including non-valid OpenAPI files or valid files that do not fulfill specific requirements and discard them. For example, it should determine and discard toy APIs at runtime.

2. The OpenAPI specification enables diverse ways to describe the same objects. Considering any possible syntax developers might use in an OpenAPI file adds unwanted complexity to the tool.

3. Concerning our goals, we still focus on the question if there are other undiscovered abstract operations. If low coverage has other reasons than missing matching units, it could wrongly indicate missing abstract operations. Assuming openISBT cannot support a service operation or even a whole document for any other reason, should the evaluation tool either consider or ignore it in the metric's calculation? For example,

   (a) openISBT does not support service operations on nested resources, also if a suitable abstract operation exists.

   (b) an OpenAPI document might define ignored service operations only. Considering the document but ignoring the service operation in the calculation leads to a paradox increase of the metric p_{APIs}. The reason for this is a division by zero : $p_i = \frac{|M_i|}{|N_i|} = \frac{0}{0} = 1$.

   (c) openISBT might throw an Exception for a certain OpenAPI document.

4. How can we systematically discard documents from our sample and keep the remaining sample representative for the statistical population?

The tool should implement the data cleaning components we describe in section 4.2 to solve these issues. Also, it should determine the distribution of abstract operations, and finally, determine the metrics $p_{operations}$ and p_{APIs} for two different cases.

For the first case, the metric should indicate missing abstract operations only, so the tool should ignore some input such as service operations on nested resources. For the second case, the metrics should determine any factor that may affect the applicability.

5.1.3 Design and Architecture

To satisfy measurement design and tools' requirements, we need a suitable tool's architecture. OpenISBT provides a command-line interface, which requires plain text files as input and

returns a text stream to the command-line's standard output, which we can pipe to other programs.

This behavior reminds of the Unix philosophy summarized by Douglas McIlroy: " Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface "[quoted in 35]. Therefore, the pipe and filter architecture is highly suitable for building a tool that wraps openISBT, and itself only handles text streams.

Figure 5.1 shows a workflow representing the tool's pipe and filter architecture. It consists of several independently executable components (also called filters) represented by the orange boxes. Each filter uses text streams as input and returns text streams as output. The solid arrows represent the connectors between two filters (also called pipes) and show the program flow. The parallelograms stand for plain text files or directories containing plain text files on the file system that store input and output data. The dashed arrows represent the data flow from a filter to a file or directory and vise versa. Some input/output is connected to multiple filters across the workflow, and for better readability, some parallelograms occur twice. The figure only shows a subset of the data flow, for instance, the entire output to the OpenAPI files directory and many less important logging files.

Components for OpenAPI Document Processing

We implement several components to handle and analyze OpenAPI documents. We require most of them to detect repetitive patterns for later investigation of the dataset. Metrics' determination requires only a single component that transforms each OpenAPI document into a shorter and more lightweight representation of the API description to speed up the calculation time. OpenAPI documents provide plenty of information about their service operations, and processing the information requires computation resources and computation time. For metrics' determination, only the URI and the HTTP method of a service operation

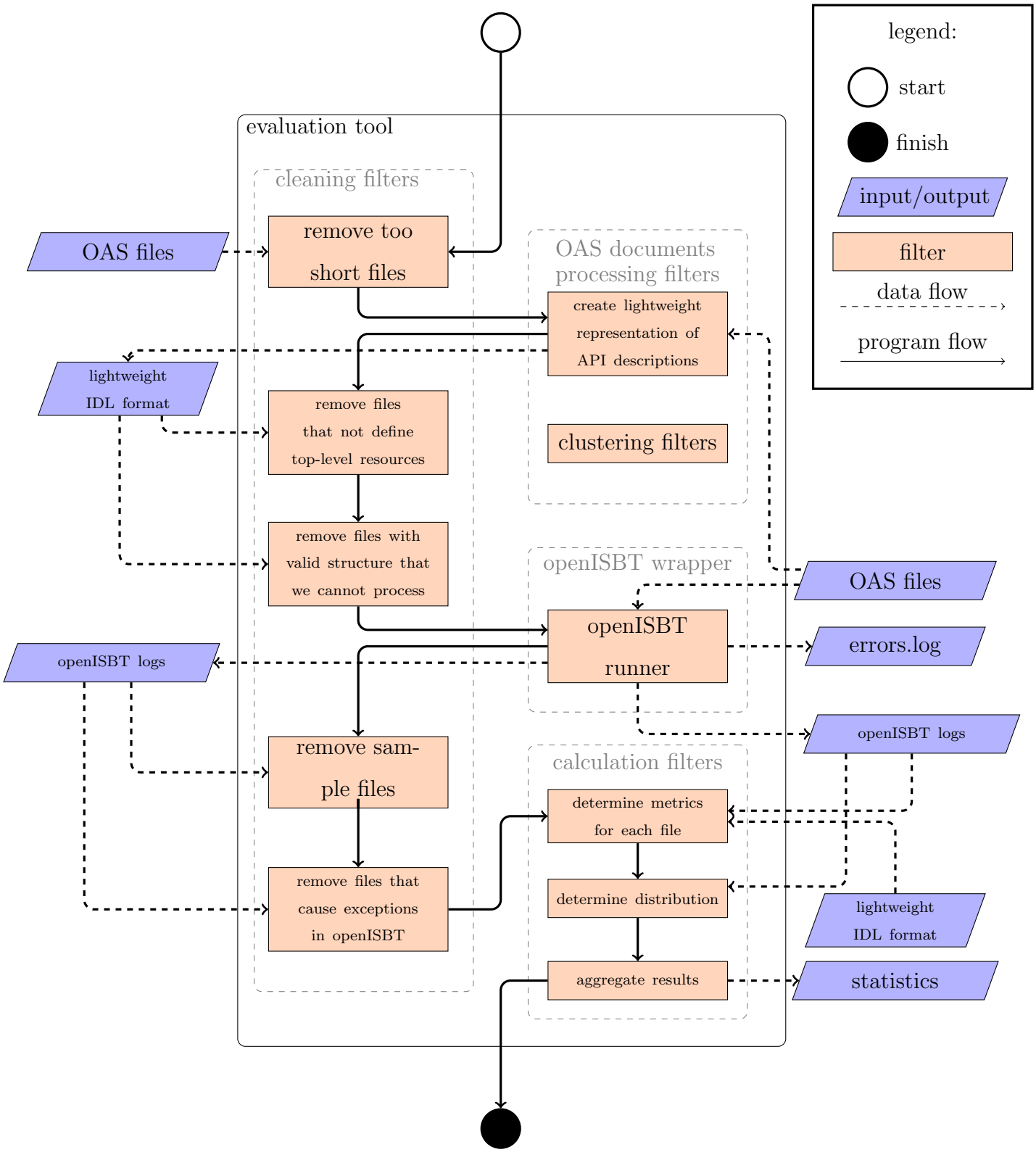


Figure 5.1: Architecture of the Evaluation Tool

are required, but we require them multiple times. Using a representation, which only contains this information, reduces the calculation time by six.

Components for Data Cleaning

Sections 4.2 and 5.1.2 discuss how each component cleans data and why. However, they do not discuss how interdependencies between components can lead to wrong results if the components' order in figure 5.1 changes. For example, assuming the API description of a toy API defines nested resources only. We first discard it as toy API, and the second cleaning component does not apply. If we first remove the document because it does not define top-level resources, we would not recognize that it is a toy API and include it in the subsequent metrics' determination. The same applies to OpenAPI documents, which openISBT cannot process without throwing exceptions.

OpenISBT Wrapper

The OpenISBT Runner is a wrapper for OpenISBT, which takes a configuration file and a set of OpenAPI files and runs the openISBT MatchingTool for each OpenAPI file. For each OpenAPI file, it redirects the output that openISBT writes to stdout into a new log file. It checks if openISBT returned output to stderr, i.e., if it threw an exception. In that case, it logs the stack trace into another log file and associates specific input files to the stack trace. Also, the openISBT Runner records its own steps and so ensures understandability.

Components for Metrics' Determination

We implement several filters for metrics' determination. First, we implement filters to determine p_i for all kept files for both cases when considering and ignoring nested resources and filters to determine metrics for files, which only define nested resources and files that cause

exceptions. Next, we implement a filter to determine the distribution of abstract operations across all service operations. Finally, we implement a filter, which aggregates the data and calculates $p_{operations}$ and p_{APIs}, as section 5.1.1 and the algorithms in Appendix A describe, and prints a summary.

5.2 Analyzing the Current State

Before we extend openISBT, we want to quantify its applicability. This section covers any aspect of the measurement execution and discusses its results.

5.2.1 Setup of the Measurement

The measurement's setup defines the required input for benchmark execution. First, it requires a specific openISBT version. For this measurement we use openISBT v0.2[1], as we want to evaluate the current state. Second, it requires a set of OpenAPI documents[2]. Finally, it requires a pattern configuration file. This file defines the abstract interaction patterns. We mention in section 2.5 that openISBT defines workload for interaction patterns, which are sequences of abstract operations. However, we require information about supported service operations, so we define in the pattern configuration for each abstract operation an abstract interaction pattern containing only this abstract operation.

5.2.2 Results of the Measurement

Before we introduce measurement results, i.e., the different metrics and the abstract operations' distribution, we introduce the sample's composition, i.e., the data cleaning results.

[1] https://github.com/martingrambow/openISBT/releases/tag/v0.2
[2] https://github.com/AmitJerochim/openapi-data

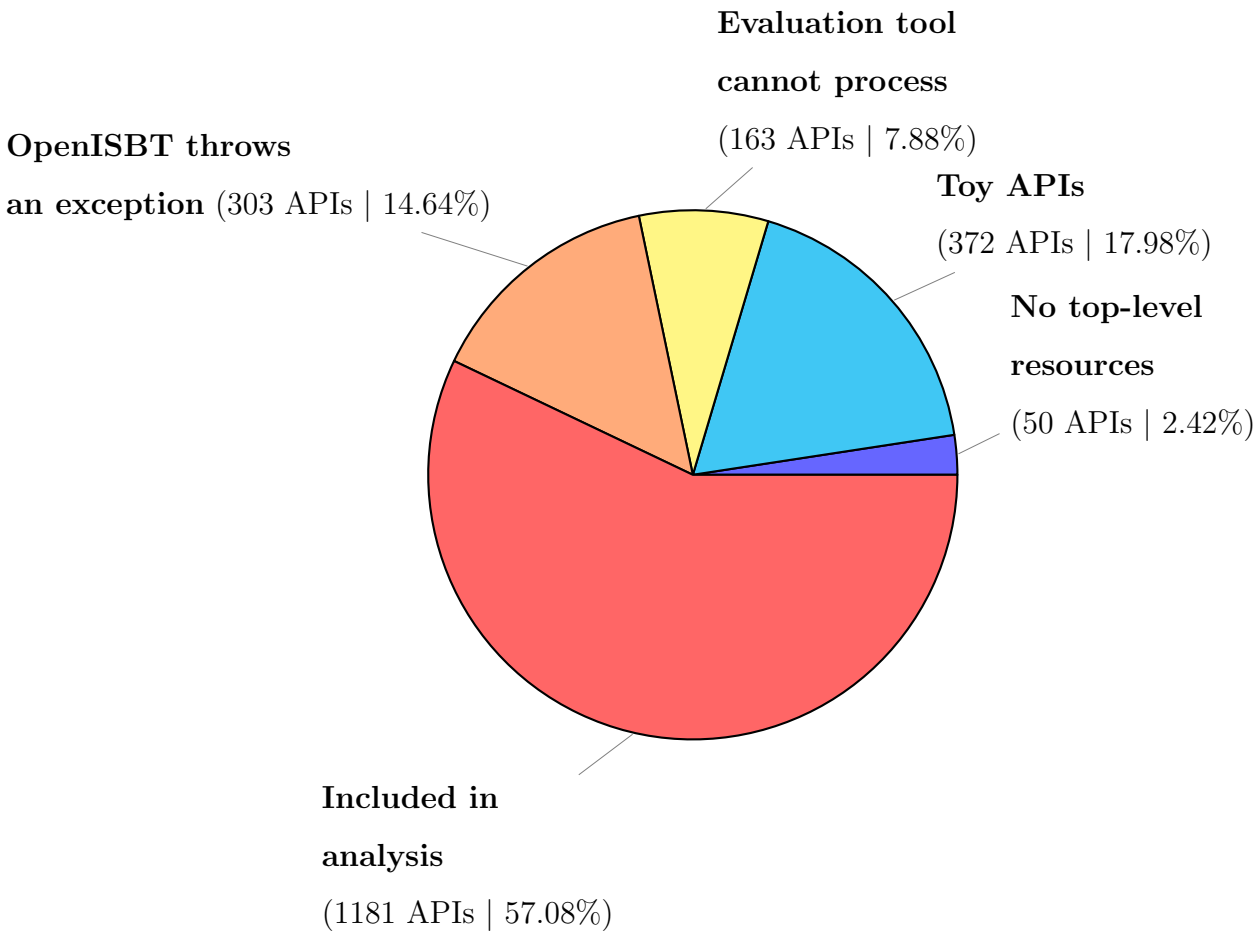
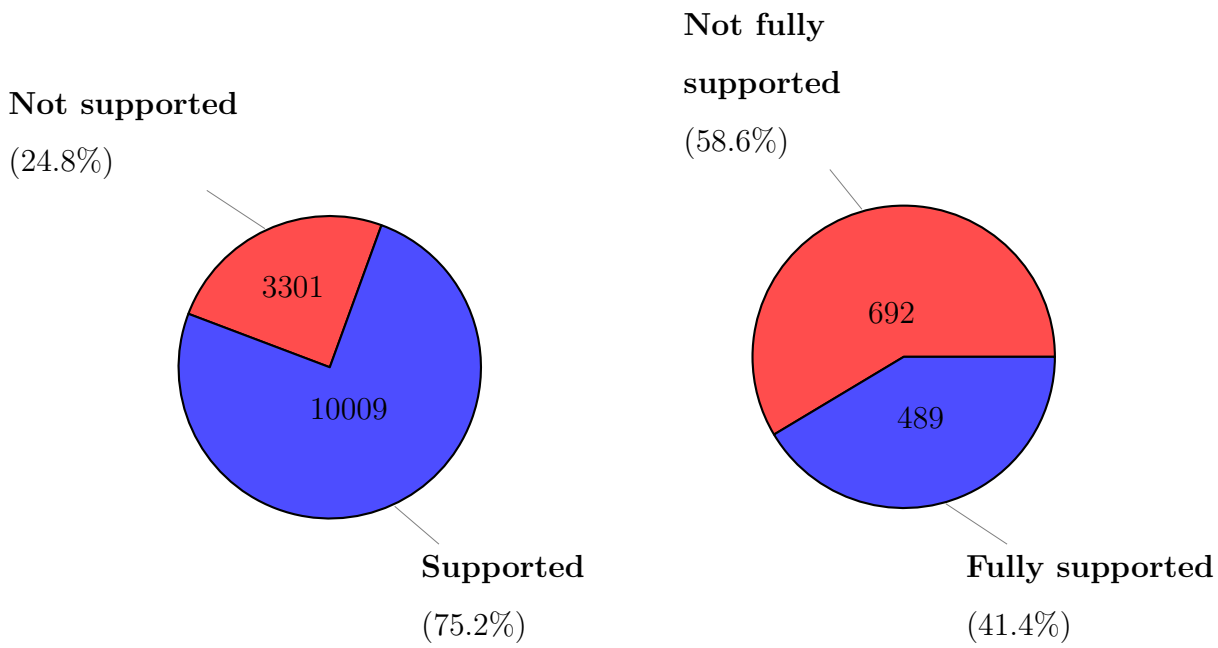


Figure 5.2: Distribution of the OpenAPI documents by applied data cleaning components

Figure 5.2 shows it as a pie chart. We pass the evaluation tool 2069 OpenAPI documents. We identify 372 OpenAPI documents, which describe toy APIs. We consider 297 of them as toy APIs as they are too short, i.e., they have less than 100 LOC. The other 75 documents are templates, which SwaggerHub provides. Also, we detect 163 OpenAPI documents, which the evaluation tool cannot process because they define general path-properties.

We also detect 303 OpenAPI documents, which openISBT cannot handle without throwing an exception. Furthermore, we detect 50 OpenAPI documents, which openISBT cannot handle since the documents only define operations on nested resources.



(a) OpenISBT matches 10009 of 13310 operations across all APIs.

(b) For 489 of 1181 APIs openISBT matches all operations on top-level resources.

Figure 5.3: Coverage metrics for a set of 1181 OpenAPI documents including 13310 service operations

We know by observation that the sample contains 158 template documents, but as we mention above, we only identify 75 template documents. Furthermore, we do not identify any document with a missing paths-property, but we know by observation that the sample contains 15 such documents. Interdependencies mentioned in section 5.1.3 are the reason for this. However, we ensure that these interdependencies do not affect results correctness.

Figure 5.3 shows two pie charts visualizing the coverage metrics. Pie chart 5.3a shows the shares of supported and unsupported service operations. Supported service operations can be successfully matched to an abstract operation by the openISBT matching tool, and unsupported operations cannot. In other words, the share of the supported operation in the pie chart is the metric $p_{operations}$. The chart mainly expresses that openISBT can match 10009 of 13310 service operations to abstract operations, i.e., the coverage $p_{operations}$ is equal to 75.2%.

Figure 5.3b shows the shares of fully supported APIs as a pie chart. Fully supported means

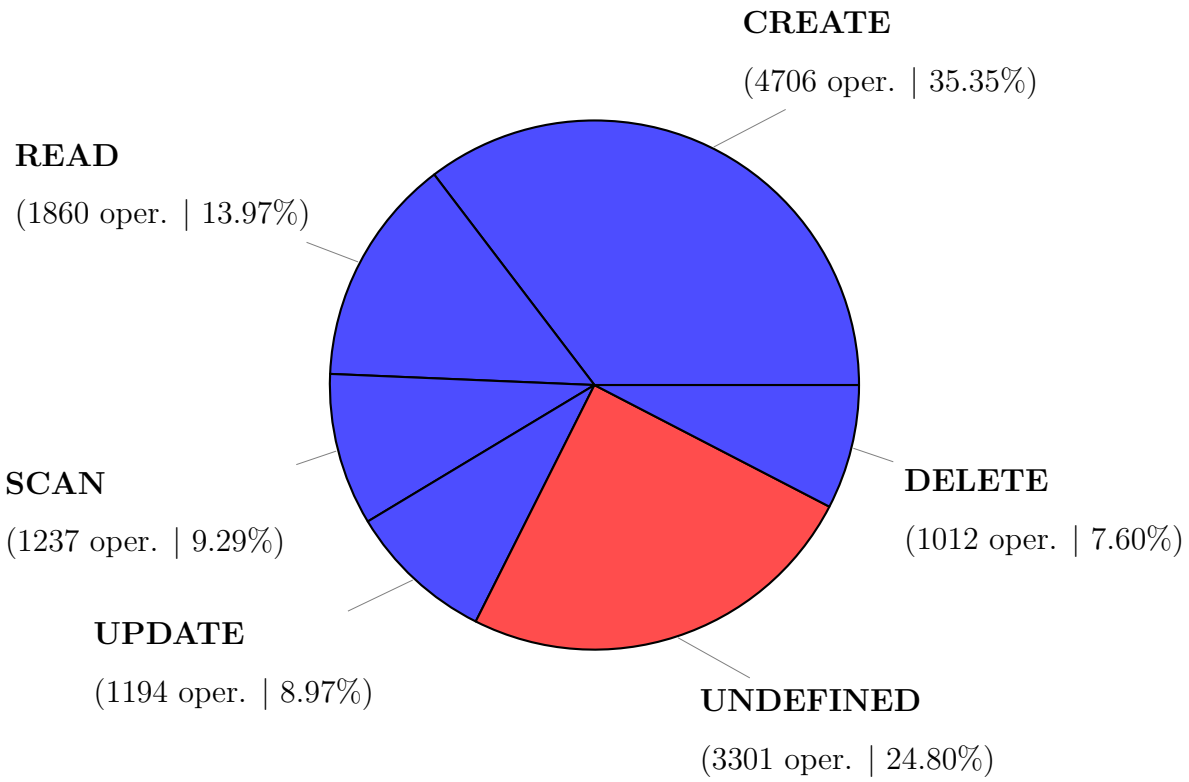


Figure 5.4: Distribution of the service operations by matched abstract operations

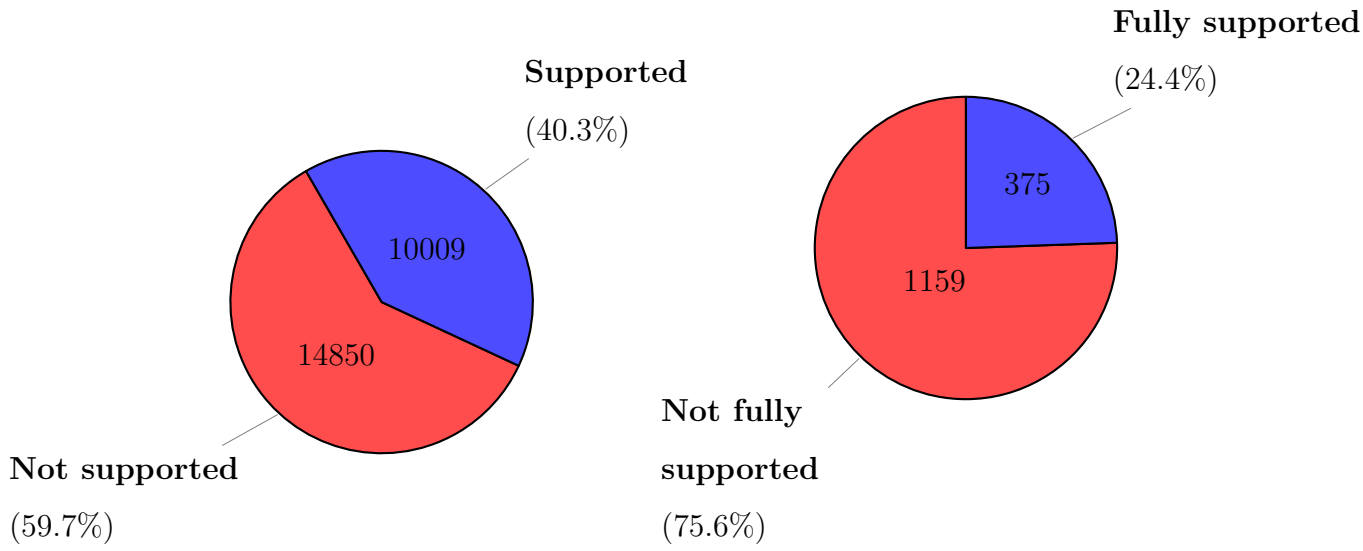
that openISBT supports all service operations an API defines. Therefore, the relative share of the fully supported APIs of 41.4% is the coverage metric p_{APIs}.

The cleaned sample of documents both pie charts refer to includes 1181 OpenAPI documents and 13310 service operations on top-level resources.

Beyond calculating coverage metrics, we also determine the distribution of the abstract operations, i.e., how often openISBT matches each abstract operation. Figure 5.4 shows the shares of each abstract operation. If a service operation is not supported, then the abstract operation is undefined yet. Therefore, the slice UNDEFINED contains all the unsupported service operations. We also notice that openISBT matches 4706 service operations (35.35%) to the abstract operation CREATE, which is significantly higher than the other operations.

5.2.3 The Larger Scope of Applicability

Determining the results above, we set focus on discovering missing abstract operations. Therefore, we ignore documents and service operations if openISBT does not support them for other reasons than undiscovered abstract operations. Figure 5.5 visualizes the metrics as figure 5.3 but reflects any aspect of applicability.



(a) OpenISBT matches 10009 of 24859 Operations across all APIs.

(b) For 375 of 1534 APIs openISBT matches all operations.

Figure 5.5: Coverage metrics for a set of 1534 OpenAPI documents including 24859 service operations

Therefore, it considers 1534 OpenAPI documents, which define 24859 service operations on top-level resources and nested resources. If openISBT cannot process an OpenAPI document without exception, or the OpenAPI document only defines nested resources, we count it as $p_i = 0$. Figure 5.5a shows a much smaller coverage of 40.3% for the supported service operations. Also, figure 5.5b shows a smaller coverage of 24.4% for the fully supported APIs.

Chapter 6

Extending OpenISBT

This chapter describes our contributions to openISBT. To contribute to openISBT, we investigate service operations that openISBT does not support and classify them based on their functionality. We implement different tools to ease the grouping, but we investigate the service operations manually. Section 6.1 discusses the general properties and requirements of abstract operations and their corresponding matching units[1]. Section 6.2 discusses findings to unsupported service operations, which should theoretically match existing abstract operations. It also explains why they do not match, and how we extended the matching units to match them. Finally, section 6.3 discusses use-cases and samples where new abstract operations and matching units are required.

[1] https://github.com/AmitJerochim/openISBT_analyser/tree/master/matching_units

6.1 General Considerations About the Design of Matching Units

A key concept of openISBT is defining workload on interaction patterns, which are sequences of abstract operations. It matches specific service operations to those abstract operations using matching units. When deriving new abstract operations and implementing matching units, there are several requirements for design and implementation. First, the abstract operations should be specific enough in order to design a realistic interaction pattern and therefore make the benchmark relevant in the meaning of [5, p. 38]. The benchmark should also be portable [5, p. 38], and the abstract operation should match to a large number of specific services. However, these are conflicting requirements, and trade-offs are crucial [5, p. 43-44]. Also, there are different implementation objectives that a matching unit has to conform to. First, the implementation should be correct in the meaning of [5, p. 38]. The abstract operations we derive are theoretical concepts. However, the matching units are responsible for matching service operations to the right abstract operations at runtime, and poorly implemented matching units can lead to wrong matching, although the abstract operation is well designed. Also, the abstract operation should be portable [5, p. 81-82] by making as few assumptions as possible. For instance, assuming that description tags in OpenAPI documents are written in English or even assuming that OpenAPI documents use description tags would lead to poor portability. Moreover, a matching unit should only consider the machine-readable elements of the API description to ensure portability.

6.2 Improving Existing Matching Units

The data investigation shows that several unsupported service operations are READ, or SCAN operations in essence. This section describes our findings and how we changed the matching units to support more service operations.

6.2.1 Improving the READ Matching Unit

READ describes an abstract operation where a client requests a single resource's representation from the server without changing its state [20, sec. 3]. Best practices for resource identification are to use path parameters with unique identifiers. The path parameter enables the implementation of a single route to request different resources of the same type,e.g., GET /users/user_id. In this example, the curly braces indicate the path parameter. This syntax is very common in many frameworks, e.g., FastAPI[2],or JAX-RS[3]. Other frameworks may use different syntax to indicate path parameters. For instance, Express.js[4] uses a colon to identify path parameters. According to the OpenAPI specification, using curly braces for path parameters is mandatory. Nonetheless, if a service provider only uses OpenAPI for documentation purposes, they frequently use colons to indicate path parameters. Even if an OpenAPI document does not entirely conform to the OpenAPI specification, openISBT can still generate a service-specific workload for the service. To increase portability, we extend the matching unit for matching service operations if a colon is used instead of curly braces in the path parameter's declaration.

6.2.2 Improving the SCAN Matching Unit

The abstract operation SCAN should map to a CRUD READ operation, which returns a list of objects [20, sec. 3]. According to the second level of RMM, a service operation, which implements a CRUD READ operation, should use HTTP GET as reading is safe and idempotent. Also, the response body should contain a collection of objects. For any service operation, we find the response body's data type in its data model (also called schema).

Listing 6.1 shows a code snippet of an exemplary OpenAPI document defining the service

[2] https://fastapi.tiangolo.com/tutorial/path-params/
[3] https://docs.jboss.org/resteasy/docs/1.0.1.GA/userguide/html/JAX-RS_Resource_Locators_and_Sub_Resources.html
[4] https://expressjs.com/en/guide/routing.html

operation GET /users. The code at lines 2-11 shows the operation's definition, and the code at lines 13-19 defines reusable components. In short, the server responds to such a request with HTTP status code 200 in the response's header and a JSON object in the response's body, which conforms to the data model (schema) in lines 8-11.

In the following, we focus on the data model. The code at line 9 defines the data type as an array. According to the OpenAPI specification, a data model of type array also requires the items property defined in lines 10-11. The $ref at line 11 refers to a *User* data model defined in lines 15-19. Line 16 defines that the *User* is of type object, and lines 17-19 defines its properties, i.e., the name property.

If the unit responsible for matching operations to SCAN inspects this service operation, it first checks if the HTTP method is GET. Then it checks if the data type at line 9 is an array. As both conditions are true, it matches the service operation to SCAN. In Listing 6.2, we rewrite the example in Listing 6.1 without changing the semantics, i.e., the snippet still describes the same operation. The code at line 9 of Listing 6.2 refers to a *Users*-schema in lines 18-21, which is an array of *User*-objects defined in lines 13-17.

The matching unit now detects *schema*-object with property *type* set to null and therefore does not match the operation to SCAN. However, it is semantically the same service operation. The OpenAPI specification enables componentization for many object types such as responses, data models, or parameters. The scenario above is manageable, but service providers use deeper nesting of components. To handle any nesting depth, we modify the matching unit to recursively resolve all references before it checks if the data type of the response is an array.

```
1          ...
2        "/users":
3         "get":
4           "responses":
5             '200':
6               "content":
7                 "application/json":
8                   "schema":
9                     "type": array
10                     "items":
11                       $ref: '#/components/schemas/User'
12        ...
13        "components":
14              "schemas":
15                "User":
16                  "type": object
17                  "properties":
18                    "name":
19                      "type": string
20        ...
```

Listing 6.1: Snippet of an OpenAPI document in YAML defining a service that maps to SCAN

```
01      ...
02      "/users":
03        "get":
04          "responses":
05            '200':
06              "content":
07                "application/json":
08                  "schema":
09                    $ref: '#/components/schemas/Users'
10        ...
11        "components":
12              "schemas":
13                "User":
14                  "type": object
15                  "properties":
16                      "name":
17                        "type": string
18                "Users":
19                  "type": array
20                  "items":
21                    $ref: '#/components/schemas/User'
22        ...
```

Listing 6.2: Snippet of an OpenAPI document in YAML defining a service that does not map to SCAN but should do.

6.3 Deriving New Abstract Operations and Implementing Matching Units

For those service operations, which openISBT does not support because current abstract operations do not describe their functions, we derive new abstract operations and implement new matching units. In the following, we present four new functions, which describe the

purposes of several service operations. We also discuss service operations' common properties, which are relevant for the matching process.

6.3.1 The CONTROL Abstract Operation

This abstract operation describes service operations where a client requests other service operation's metadata. The use cases for such an operation vary, so we present different examples. The first example assumes that a client considers re-downloading a large resource. Using a CONTROL operation, it could check if the resource has been modified since the last download. In scenarios where bandwidth is limited, the client might also check the resource's content-length before requesting it.

In some scenarios, clients require information about accepted requests. To receive this information, the client would perform a preflight request to the server. For most preflight requests, developers do not need to implement the service operations explicitly. For example, if a client uses a preflight request to ask the server if it would allow an HTTP DELETE request. For some preflight requests, explicitly defining and implementing the service operation is required. For example, if a client sends a cross-origin request including cookies or non-standard headers to domain A from a script served by domain B.

A client needs to perform an HTTP HEAD request to only ask for the resource's metadata. Browsers use the HTTP OPTIONS method to perform preflight requests. Both HTTP methods are safe and idempotent and allow neither request body nor response body.

6.3.2 The AUTHENTICATE Abstract Operation

Authentication is the act where an entity, e.g., a person or a machine, proves its identity to another entity. We notice that common authentication mechanisms in the WWW follow specific standards, such as [1, 13, 24, 25, 34]. Since HTTP and REST are stateless, the client

authenticates itself with each request. Defining an abstract operation only for authentication seems questionable. For example, a service operation, which matches SCAN, might require an authorization header field [28]. However, the authorization header contains credentials [13], e.g., a JWT token [24], and clients require service operations to request credentials, e.g., GET /login.

We implement a matching unit, which checks if the response body contains an access token. Also, the service operation should use HTTP GET, as the request should not change the server's state.

6.3.3 The VALIDATE Abstract Operation

In some cases, services should fulfill validation tasks, e.g., validating the correctness of a phone number or a bank account. Validation can be described by a function that accepts any input and returns a true or false.

However, finding a common pattern for such services is challenging. First, assuming an HTTP request includes data to validate, we still do not clearly know where to find them. For instance, the service operation GET /vouchers/validate[5] expects the data to be in the URI. In contrast, POST /api/Order/validate[6] expects the data to be in the request body. Hybrid variants also exist. For instance, POST /edifact/validate[7] expects data in the request body and additional metadata in the URI.

Next, we assume that the response should be a truth-value, which is a resource in the meaning of REST in a broader sense. We mention in 2.3 that resources are abstract concepts and might have different representations. For example, we can use binary numbers to represent

[5] https://github.com/AmitJerochim/openapi-data/blob/master/oasFiles/0254_bigdish-consumer-app-1.0.8-swagger.json
[6] https://github.com/AmitJerochim/openapi-data/blob/master/0187_AspenwareUnity-UnityAPI-3.0-oas3-swagger.json
[7] https://github.com/AmitJerochim/openapi-data/blob/master/0604_EdiNation-edi-nation-api-2-swagger.json

truth-values. Strings are also a suitable representation , e.g. "valid" and "invalid", "correct" and "not correct", or "true" and "false". JSON and XML fit as well, e.g., {"*valid* : "*true*"}, < *valid* > *true* < /*valid* >. We also might use HTTP status codes to represent truth-values., e.g., 200 for true and 400 for false. Finally, hybrid variants exist as well.

Indeed, we find several representatives for the examples above. For instance, POST /remittance/validate[8] might answer with {"*status* : "*failed*"}, while POST /api/Order/validate[9] either respond with "true" or "false". The service operation POST /Order/validate[10] uses JSON and XML and POST /admin/schema/validate[11], or POST /x12/validate[12] use both HTTP status codes.

A matching unit, which considers any case, would match many wrong service operations. We design a matching unit that only matches service operation if they respond with a JSON object containing a boolean. We also present several service operations that use HTTP POST. However, we assume that validation should not change the server's state. Furthermore, we consider it is idempotent since multiple repeats of the validation process for a specific input should not return in different validation states. Therefore, the matching unit only matches service operations that use HTTP GET.

6.3.4 The INFO Abstract Operation

This abstract operation describes service operations, which respond with non-persisted data. For example, users' activity status in messaging applications, or social networks, is often

[8] https://github.com/AmitJerochim/openapi-data/blob/master/oasFiles/2024_zheiro-ubx-i2i-api-portal-0.2.2-swagger.json
[9] https://github.com/AmitJerochim/openapi-data/blob/master/0187_AspenwareUnity-UnityAPI-3.0-oas3-swagger.json
[10] https://github.com/AmitJerochim/openapi-data/blob/master/oasFiles/1708_Someorg1-au2web-v3-oas3-swagger.json
[11] https://github.com/AmitJerochim/openapi-data/blob/master/oasFiles/1384_osroot-yuuvis-1.0-swagger.json
[12] https://github.com/AmitJerochim/openapi-data/blob/master/0604_EdiNation-edi-nation-api-2-swagger.json

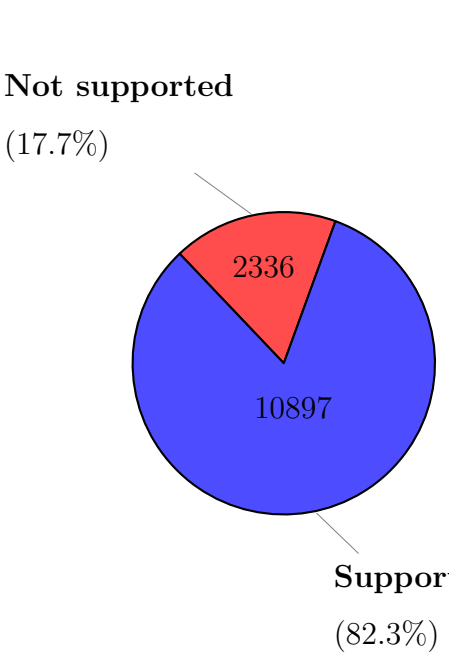
visible to other users in real-time.

A service for retrieving information, persisted or not, should be safe and idempotent. Therefore, we presume the HTTP method to be GET. Additionally, a matching unit should distinguish between persisted and non-persisted service operations. Therefore, we assume that developers use meaningful URIs and match service operations based on the URI semantics. For the sample above, checking the activity of a user could be done by performing GET /users/{id}/status where {id} is a placeholder for a specific user-id.

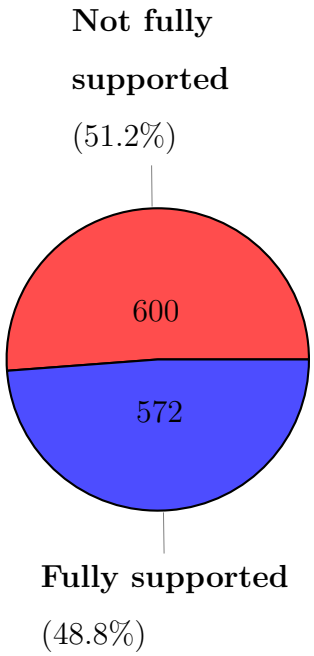
Chapter 7

Evaluating New Abstract Operations

This chapter covers aspects of the second measurement execution for the extended openISBT version. It presents the results and compares them to the results of the previous measurement.



(a) OpenISBT matches 10897 of 13233 operations across all APIs.



(b) For 572 of 1172 APIs openISBT matches all service operations on top-level resources.

Figure 7.1: Coverage metrics determined in the second measurement

We use an extended pattern configuration file for the measurement execution, which defines four new interaction patterns, each containing one new abstract operation. Appendix A.2 shows that nine more OpenAPI documents cause exceptions in openISBT because of code changes. Therefore, we determine metrics for a smaller sample and can only compare relative values.

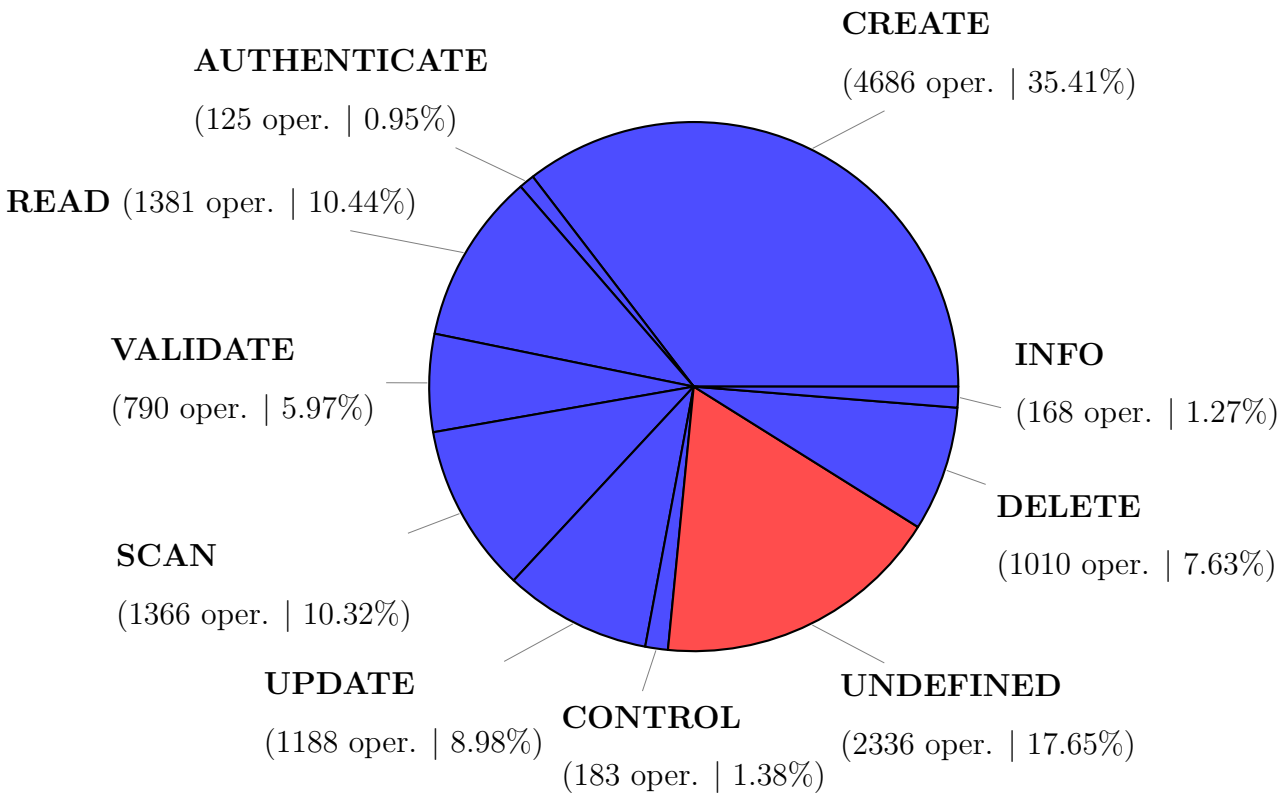


Figure 7.2: Distribution of the service operations by matched abstract operations for the second measurement

The following metrics belong to a set of 1172 OpenAPI documents, which define 13233 service operations. Figure 7.1 shows two pie charts, which visualize the coverage metrics $p_{operations}$ and p_{APIs}. Figure 7.1a shows that the extended openISBT version can match 10897 (82.3%) service operations. Therefore, the coverage $p_{operations}$ is 7.1% higher than in the first measurement execution (see figure 5.3a).Figure 7.1b shows that the extended openISBT version can match any operation in 572 (48.8%) OpenAPI documents. Comparing it to figure 5.3b, we notice an increase of p_{APIs} of 7.4%.

Figure 7.2 shows the distribution of abstract operations as a pie chart. Comparing it to the prior distribution in figure 5.4, we see that each value changed. However, for the abstract operations CREATE, UPDATE, and DELETE, the change is minimal and subject to the changed sample size. The decrease in the coverage of undefined operation is $\Delta p_{operations}$. There are representants for all new abstract operations in the sample. We see an increase in the coverage of the SCAN operation from 9.29% to 10.32%. The coverage of the READ operation shows a decrease from 13.97% to 10.44%. As the modifications to the matching units we describe in section 6.2 are not restrictive, we assume a shift from the READ operation to other abstract operations.

Chapter 8

Discussion

This work shows that openISBT can benchmark a broad set of service operations and entire APIs. It also shows how we extend openISBT and so increase portability. However, there are some points to consider about our experiment's methodology, the pattern-based approach, the design and implementation of openISBT, and our contributions to openISBT.

8.1 Considerations About the Methodology of the Experiment

This section reviews our experiment's methodology and design and discusses critical decisions and assumptions we make in this work.

Assumptions about toy APIs We remove 372 OpenAPI documents from the sample because they describe toy APIs. We know that 75 OpenAPI documents indeed describe toy APIs, but for the other 297, we only know that they have less than 100 LOC. We assume that service providers do not use interface descriptions with few details in real-world applications

and risk removing required documents and keeping unwanted documents.

Assumption about OpenAPI documents that cause exceptions Section 5.2.3 discusses the impact of poor exception handling in openISBT on portability and presents a significantly lower coverage of supported service operations. We consider all service operations of an OpenAPI document as unsupported if openISBT throws an exception during this document's processing. However, we know by observation that OpenAPI documents, which cause exceptions, indeed define supported operations. Nonetheless, counting supported operations in metrics calculation is not feasible. OpenISBT stops working if an exception is thrown. Therefore, the log files do not show eventually supported service operations.

Comparison of ratios from different subsets of the sample If openISBT throws an exception during the processing of an OpenAPI document, we remove this document. As different versions of openISBT might throw different exceptions, the subset of removed documents differs for each version's measurement. Therefore, we compare metrics for samples of different sizes. As we only compare relative means, we rely on the central limit theorem and reasonably large sample sizes [22].

8.2 Assumptions of the Authors of the Pattern-based Benchmarking Approach

The authors rely on the second level of Richardson's maturity model, which restricts the HTTP methods of a service operation to be as close as possible to how it is used in HTTP itself. By observation, we know that openISBT sometimes matches service operations to wrong abstract operations (false-positive errors) because their APIs do not follow the second level of Richardson's maturity model. We also know by observation that false-positive errors

occur for most abstract operations occasionally. For CREATE, we observe a significantly higher rate.

To quantify the observation, we design and implement an assumption-based detection tool. We estimate 957 service operations that openISBT matches wrongly to CREATE. We discuss these assumptions and the results in details in appendix A.3.

8.3 Limitations of the OpenISBT Implementation

The openISBT implementation has issues that the authors' approach does not affect, so we discuss the prototype's limitations independently of the approach's limitations. Section 5.2.3 already discusses the implications of nested resources in OpenAPI documents and poor exception handling. In our work, we notice a third limitation regarding the openISBT implementation. The approach assumes that each service operation is matched to exactly one abstract operation, but we notice that a service operation can be matched to multiple abstract operations. The authors fix this issue by prioritizing abstract operations. For instance, SCAN is more specific than READ and therefore has a higher priority. This approach works well for many service operations but might also lead to false-positive errors. Our modifications to the matching units in section 6.2 solve this issue for SCAN and READ without prioritization by checking mutually exclusive properties. We cannot find mutually exclusive properties for the abstract operations VALIDATE, and READ, so we prioritize VALIDATE over READ. However, prioritizing VALIDATE might be a source of false-positive errors because openISBT might match service operations to VALIDATE, although they should match them to READ.

8.4 Outlook and Future Research

OpenISBT is subject to continuous changes. Our work shows that unwanted behavior occurs because of the lack of test cases (see sections 5.2.3, 6.2, 8.3). The evaluation tool we present in section 5.1 cannot find any unexpected behavior, but associates caught exceptions to OpenAPI documents. Furthermore, it aggregates them and allows developers to find issues in the code so that future contributors can use the tool in the development process.

We also show that openISBT matches service operations wrongly if they do not follow the second level of Richardson's maturity model. Therefore, we use an assumption-based detection tool to estimate false-positive errors. Future researchers could determine the exact ratio of false-positive ratios and either re-implement the CREATE matching unit or propose an approach to detect Richardson's maturity level of OpenAPI documents.

Chapter 9

Related Work

In this work, we evaluate the applicability of openISBT, a proof-of-concept prototype of the pattern-based benchmarking approach, and extend its applicability. This chapter compares the pattern-based benchmarking approach to prior results in this field. IT-benchmarking is a broad area and covers different fields such as hardware benchmarking or software benchmarking, and each of these fields is broad itself. Both fields can easily overlap. For example, a common microservice might include a data storage system and run in a container, which runs on a cluster. Therefore, a clear separation of all kinds of SUT is difficult, so we consider more kinds of systems than just microservices in this section. Also, there are benchmarks for measuring several non-functional properties, and to narrow down our research field by the SUT is not necessarily the best approach. For example, a security benchmark for DBMS and a security benchmark for REST APIs might have more in common than a security benchmark and a performance benchmark for REST APIs.

Therefore, this section discusses several fields of research. There are different virtual machines and containers benchmarking approaches [6, 8, 40–42] that introduce powerful tools for measuring different virtualization scenarios. There are several approaches for data storage system benchmarking [2, 10, 11, 27, 32, 33, 43], web-application benchmarking [3, 4], and

also application-driven benchmarking approaches [18, 23, 37]. However, openISBT appears to be the only approach that provides a generic benchmark for microservices.

Similar to our approach, Zheng et al. also use interaction patterns[43]. Their approach is only designed for Cloud Object Storage. Therefore, it relies on CDMI, an interface based on RESTful principles but is less heterogeneous than REST.

Steven Bucaille et al. use OpenAPI to measure non-functional properties of REST APIs [7]. In particular, they measure the performance and availability of services in different geographical locations by means of a master/slave architecture. However, their approach focuses on the monitoring of REST APIs and not on benchmarking. Also, their approach is not pattern-based but handles each operation the same way. Stefan Karlsson et al. use OpenAPI to dynamically generate tests for functional properties [26] and run them against the implementation from a client's perspective. Their research does not belong to the area of IT-benchmarking or even the measurement of non-functional properties of systems. However, they discuss the difficulty of resolving parameters under the term "stateful sequences" and implicitly use interaction patterns.

Section 2.1 mentions RabbitMQ and ZeroMQ as alternatives to REST. Currently, openISBT can not benchmark microservices using these mechanisms, which is related to the fact that the OpenAPI specification is not a proper IDL for them. However, Xian Jun Hong et al. compare REST-based and RabbitMQ-based microservices and show that for a high workload, asynchronous mechanisms have an advantage over REST APIs [21]. OpenAPI evolved in the last decade to a de facto standard among the RESTish IDLs and is discussed in several academic papers. At the same time, asynchronous APIs such as RabbitMQ and ZeroMQ had neither an IDL nor all products built around it (e.g., stub generator). AsyncAPI[1] is a new machine-readable interface description language for asynchronous APIs that reached academia in the year 2020. To the best of our knowledge, there is only one academic paper where Abel Gómez et al. evaluate the usage of AsyncAPI for application development pur-

[1] https://www.asyncapi.com/

poses [19]. Nonetheless, AsyncAPI allows usage of YAML and JSON, as OpenAPI does. Both languages' syntax is very similar, which reduces the effort to extend openISBT to benchmark event-driven and asynchronous microservices.

Chapter 10

Conclusion

Benchmarking serves to improve the non-functional properties of IT-systems. Designing a portable benchmark for microservices is difficult as they do not share a common interface. Martin Grambow et al. present a pattern-based approach for defining portable and relevant microservices' benchmarks. Their approach defines abstract workload on interaction patterns, i.e., sequences of abstract operations, and maps service operations to generate service-specific workload. They present openISBT, a proof-of-concept prototype that is only evaluated against a few toy services.

In this work, we evaluated the applicability of openISBT for a large set of open source microservices APIs. We extended its applicability by deriving new abstract operations and implementing matching units. In particular, we implemented a web crawler, which automatically collects OpenAPI documents from swagggerhub.com and collected 2069 OpenAPI documents. Also, we derived metrics to quantify applicability and designed and implemented an evaluation tool to measure these metrics for openISBT. Finally, we derived four abstract operations, implemented new matching units, and modified existing matching units. Our contributions to openISBT increased the coverage of fully supported APIs from 41.4% to 48.8% and the coverage of supported service operations from 75.2% to 82.3%.

We also discussed different limitations and stated future directions for research. First, we showed that several APIs do not follow the second level of Richardson's maturity model, so openISBT matches their service operations to wrong abstract operations. For instance, we found 957 possible service operations that do not use HTTP POST in a RESTful way. To reduce errors, reliable automation for detecting non-REST API is crucial. We also showed that the prioritization of abstract operations leads to errors. Matching units should rather rely on mutually exclusive properties of service operations. Next, openISBT only benchmarks REST API but not asynchronous APIs. We assume that extending openISBT for asynchronous APIs is possible with less effort by using AsyncAPI as IDL. Finally, we showed that malfunction occurs because of a lack of test cases and provided open source tools and data to improve the development process.

Appendix A

Technologies and Concepts of a Larger Context

A.1 Algorithms for the Calculation of Metrics

Section 5.1.1 introduces new coverage metrics for applicability measurement. This section presents algorithms for their calculation. We use algorithm 2 to calculate the coverage metric p_{APIs} and algorithm 3 to calculate $p_{operations}$.

In algorithm 2, we initialize the counter SUPPORTED_APIS to zero. Then we iterate through all OpenAPI files and check the condition $p_i = 1$. If it evaluates to true, then we increment SUPPORTED_APIS. After the last iteration, we determine and return the quotient between SUPPORTED_APIS and ALL_APIS as p_{APIs}.

Algorithm 2: Calculate p_{APIs}

input : set of OpenAPI files

input : pattern configuration file

output: p_{APIs}

ALL_APIS = number of OpenAPI files;

SUPPORTED_APIS = 0;

for *each OpenAPI file* **do**

    p_i = calculate p_i ;

    if p_i = *1* **then**

        increment SUPPORTED_APIS ;

p_{APIs} = SUPPORTED_APIS divided by ALL_APIS ;

Algorithm 3 iterates over all OpenAPI files as well. It initializes the variable SUM to zero and adds all values of p_i for all OpenAPI files to SUM. Finally, it divides SUM by the number of OpenAPI documents and returns it.

Algorithm 3: Calculate $p_{operations}$

input : set of OpenAPI files

input : pattern configuration file

output: $p_{operations}$

ALL_APIS = number of OpenAPI files;

SUM = 0;

for *each OpenAPI file* **do**

    p_i = calculate p_i ;

    SUM = SUM + p_i ;

$p_{operations}$ = SUM divided by ALL_APIS ;

A.2 Results of the Data Cleaning for the Second Measurement Execution

The evaluation tool we present in section 5.1 removes OpenAPI documents for each measurement execution. In the second measurement execution, we use the same sample of OpenAPI documents as the first measurement execution but another openISBT version. Figure A.1 shows the frequency distribution of the processed OpenAPI documents for the second measurement execution. It is almost the same as in figure 5.2 but shows a minimal increase of OpenAPI documents that cause exceptions.

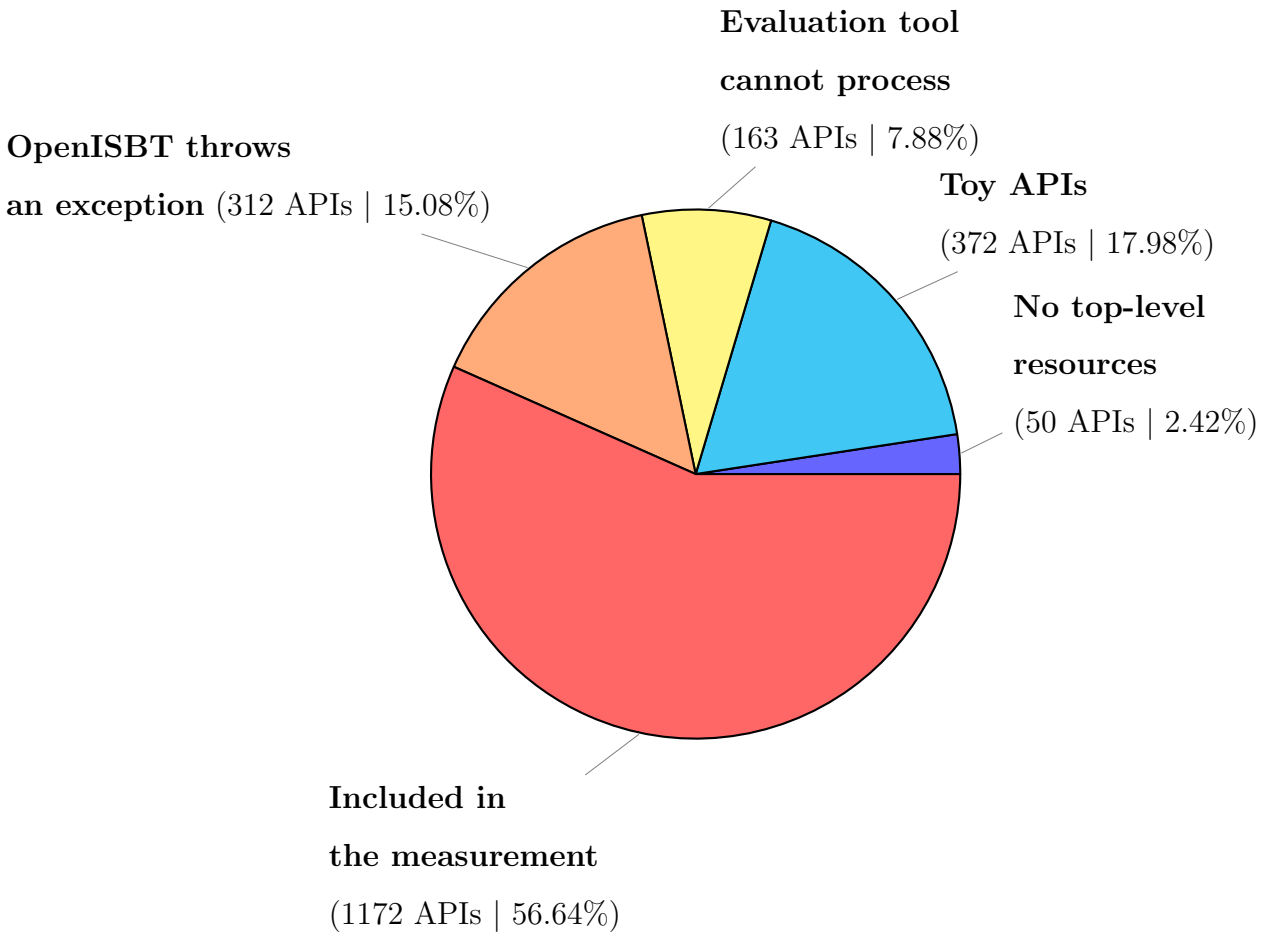


Figure A.1: Distribution of the OpenAPI documents by applied data cleaning components for the second measurement execution

A.3 Detection of False-positive Errors of the CREATE Matching Unit

To determine the number of false-positive errors, we implement an assumption-based detection tool[1]. Detecting these errors is difficult. Many OpenAPI documents include various languages in descriptions fields, and many others even do not contain any description. Next, openISBT produces false-positive errors because we make assumptions when designing matching units. Making more assumptions to detect these errors probably leads to new errors. First, we might wrongly detect true-positive results, and second, we might miss false-positive errors. However, we only use this tool to estimate the number of false-positive errors.

First, we assume that to create a new resource, the client needs to provide a representation of the resource in the request body. In reverse, a request without a request body is probably used as a READ, SCAN, or AUTHENTICATE. The tool detects 607 service operations that do not require a request body. OpenISBT matches many of them wrongly (e.g., POST /getBanks[2]). However, the tool also detects correctly matched operations (e.g., POST /student/post[3]). It also does not detect all false-positive errors (e.g, POST /getVehicleOwnerContactInfo[4]).

For the remaining service operations with an explicitly defined request body, we check if the substring "update" appears in the operation definition. We determine 316 services, and most of them are used to update resources (e.g., POST /customer/update[5]). We also find a few correctly matched service operations (e.g., POST /tasks[6]).

[1] https://github.com/AmitJerochim/openISBT_analyser/blob/master/clustering/requestbody_checker_iterator.sh
[2] https://github.com/AmitJerochim/openapi-data/blob/master/oasFiles/2018_ZAR-Network-api.zar.cloud-0.1.0-swagger.json
[3] https://github.com/AmitJerochim/openapi-data/blob/master/oasFiles/1542_Repatik-Repatik-1.0.0-oas3-swagger.json
[4] https://github.com/AmitJerochim/openapi-data/blob/master/oasFiles/1666_shivek-automation-rgm-buslane-demo-1.0.0-swagger.json
[5] https://github.com/AmitJerochim/openapi-data/blob/master/oasFiles/1902_vesteliotcloudteam-NativeApi-0.0.11.2-swagger.json
[6] https://github.com/AmitJerochim/openapi-data/blob/master/oasFiles/2027_ZID-egov-fa-services-0.0.1-swagger.json

Using the same approach but searching for "delete" instead of "update", we determine 40 possible service operations. We know by observation that exactly 34 of them semantically should be matched to DELETE.

Appendix B

Acronyms

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

CI Continuous Integration

CMDI Cloud Data Management Interface

CRUD Create, Read, Update, Delete

CSS Cascading Style Sheets

DBMS Database Management System

DOM Document Object Model

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IDL Interface Description Language

JSON JavaScript Object Notation

LOC Lines of Code

OAS OpenAPI Specification

REST Representational State Transfer

RMM Richardson's maturity model

SaaS Software as a Service

SUT System Under Test

URI Uniform Resource Identifier

URL Uniform Resource Locator

UTF Unicode Transformation Format

YAML YAML Ain't Markup Language

YCSB Yahoo! Cloud Serving Benchmark

Bibliography

[1] A. Barth. *HTTP State Management Mechanism*. RFC 6265. Apr. 2011.

[2] David Bermbach, Jörn Kuhlenkamp, Akon Dey, Arunmoezhi Ramachandran, Alan D. Fekete, and Stefan Tai. "BenchFoundry: A Benchmarking Framework for Cloud Storage Services". In: *Service-Oriented Computing - 15th International Conference, ICSOC 2017, Malaga, Spain, November 13-16, 2017, Proceedings*.

[3] David Bermbach and Erik Wittern. "Benchmarking Web API Quality". In: *Web Engineering - 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings*. Springer.

[4] David Bermbach and Erik Wittern. "Benchmarking Web API Quality - Revisited". In: *CoRR* (2019).

[5] David Bermbach, Erik Wittern, and Stefan Tai. *Cloud Service Benchmarking - Measuring Quality of Cloud Services from a Client Perspective*. Springer, 2017.

[6] Amir Hossein Borhani, Philipp Leitner, Bu-Sung Lee, Xiaorong Li, and Terence Hung. "WPress: An Application-Driven Performance Benchmark for Cloud-Based Virtual Machines". In: *18th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2014, Ulm, Germany, September 1-5, 2014*.

[7] Steven Bucaille, Javier Luis Cánovas Izquierdo, Hamza Ed-Douibi, and Jordi Cabot. "An OpenAPI-Based Testing Framework to Monitor Non-functional Properties of REST

APIs". In: *Web Engineering - 20th International Conference, ICWE 2020, Helsinki, Finland, June 9-12, 2020, Proceedings*. Springer, 2020.

[8] Francisco Carpio, Marta Delgado, and Admela Jukan. "Engineering and Experimentally Benchmarking a Container-based Edge Computing System". In: (2020).

[9] David A. Chappell. *Enterprise service bus - theory in practice*. O'Reilly, 2004.

[10] Akon Dey, Alan D. Fekete, Raghunath Nambiar, and Uwe Röhm. "YCSB+T: Benchmarking web-scale transactional databases". In: *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*.

[11] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases". In: *Proc. VLDB Endow.* ().

[13] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Authentication*. RFC 7235. June 2014.

[15] Roy Thomas Fielding. "REST: Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine, 2000.

[18] Yu Gan et al. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM.

[19] Abel Gómez, Markel Iglesias-Urkia, Aitor Urbieta, and Jordi Cabot. "A model-based approach for developing event-driven architectures with AsyncAPI". In: ACM, 2020.

[20] Martin Grambow, Lukas Meusel, Erik Wittern, and David Bermbach. "Benchmarking Microservice Performance: A Pattern-based Approach". In: *Proceedings of the 35th ACM Symposium on Applied Computing (SAC 2020)*. ACM.

[21] Xian Jun Hong, Hyun Sik Yang, and Young Han Kim. "Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application". In: *International Conference on Information and Communication Technology Convergence, ICTC 2018, Jeju Island, Korea (South), October 17-19, 2018*. IEEE.

[22] Mohammad Islaqm. "Sample size and its role in Central Limit Theorem (CLT)". In: *Computational and Applied Mathematics Journal* (2018).

[23] Devki Nandan Jha, Zhenyu Wen, Yinhao Li, Michael Nee, Maciej Koutny, and Rajiv Ranjan. "A Cost-Efficient Multi-cloud Orchestrator for Benchmarking Containerized Web-Applications". In: *Web Information Systems Engineering - WISE 2019 - 20th International Conference, Hong Kong, China, November 26-30, 2019, Proceedings*.

[24] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015.

[25] M. Jones and D. Hardt. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. RFC 6750. Oct. 2012.

[26] Stefan Karlsson, Adnan Causevic, and Daniel Sundmark. "QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs". In: *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020.

[27] Markus Klems, David Bermbach, and Rene Weinert. "A Runtime Quality Measurement Framework for Cloud Database Service Systems". In: *8th International Conference on the Quality of Information and Communications Technology, QUATIC 2012, Lisbon, Portugal, 2-6 September 2012, Proceedings*.

[32] Ioannis Papapanagiotou and Vinay Chella. "NDBench: Benchmarking Microservices at Scale". In: *CoRR* (2018).

[33] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio César López-Hernández, Garth Gibson, Adam Fuchs, and Billie Rinaldi. "YCSB++: benchmarking

and performance debugging advanced features in scalable table stores". In: *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*. ACM.

[34] J. Reschke. *The 'Basic' HTTP Authentication Scheme*. RFC 7617. Sept. 2015.

[35] Peter H. Salus. *A quarter century of UNIX*. Addison-Wesley, 1994.

[37] Akshitha Sriraman and Thomas F. Wenisch. "μ Suite: A Benchmark Suite for Microservices". In: *2018 IEEE International Symposium on Workload Characterization, IISWC 2018, Raleigh, NC, USA, September 30 - October 2, 2018*. 2018.

[40] Blesson Varghese, Lawan Thamsuhang Subba, Long Thai, and Adam Barker. "Container-Based Cloud Virtual Machine Benchmarking". In: *CoRR* (2016).

[41] Kejiang Ye, Zhaohui Wu, Bing Bing Zhou, Xiaohong Jiang, Chen Wang, and Albert Y. Zomaya. "Virt-B: Towards Performance Benchmarking of Virtual Machine Systems". In: *IEEE Internet Comput.* ().

[42] Dmitry Zaitsev and Piotr Luszczek. "Docker container based PaaS cloud computing comprehensive benchmarks using LAPACK". In: *Proceedings of The Third International Workshop on Computer Modeling and Intelligent Systems (CMIS-2020), Zaporizhzhia, Ukraine, April 27-May 1, 2020*.

[43] Qing Zheng, Haopeng Chen, Yaguang Wang, Jiangang Duan, and Zhiteng Huang. "COSBench: A Benchmark Tool for Cloud Object Storage Services". In: *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*.

Bibliography (Online)

[12] Omar Elgabrys. *The Ultimate Guide to Data Cleaning*. 2019. URL: https://towardsdatascience.com/the-ultimate-guide-to-data-cleaning-3969843991d4.

[14] Roy Thomas Fielding. *REST APIs Must Be Hypertext-driven*. 2008. URL: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

[16] Martin Fowler. *Microservices - a definition of this new architectural term*. 2014. URL: https://martinfowler.com/articles/microservices.html.

[17] Martin Fowler. *Richardson Maturity Model-Steps Toward The Glory Of REST*. Mar. 18, 2013. URL: https://martinfowler.com/articles/richardsonMaturityModel.html.

[28] *OpenAPI 3.0 specification, Section Authentication and Authorization*. Smartbear. URL: https://swagger.io/docs/specification/authentication.

[29] *OpenAPI 3.0 specification, Section Operation Object*. Smartbear. URL: https://swagger.io/specification/?_ga=2.188531752.1882554254.1605818633-2140793719.1604267198%5C#operation-object.

[30] *OpenAPI 3.0 specification, Section Path Item Object*. Smartbear. URL: https://swagger.io/specification/?_ga=2.188531752.1882554254.1605818633-2140793719.1604267198%5C#path-item-object.

[31] *OpenAPI 3.0 specification, Section Specification Extensions*. Smartbear. URL: https://swagger.io/specification/?_ga=2.188531752.1882554254.1605818633-2140793719.1604267198%5C#specification-extensions.

[36] Kristopher Sandoval. *AsyncAPI: 2020's Industry Standard For Messaging APIs?* 2020. URL: https://nordicapis.com/asyncapi-2020s-industry-standard-for-messaging-apis/.

[38] Guido Steinacker. *Why Microservices?* OTTO. URL: https://www.otto.de/jobs/technology/techblog/artikel/why-microservices_2016-03-20.php.

[39] *SwaggerHub Documentation For Searching Documents*. Smartbear. URL: https://app.swaggerhub.com/help/ui/searching.